Faculdade de Engenharia da Universidade do Porto

I2C Network for Home Automation

João Pedro Sousa Monteiro

Mestrado Integrado
em
Engenharia Electrotécnica e de Computadores

External Supervisor: João Manuel Moura Paredes

Internal Supervisor: João Paulo Filipe de Sousa

July 28, 2017

Abstract

The aim of the present document is to report the research and development of a automatically configuring I2C framework in the context of a home automation system, aiming to provide an affordable, secure and easy to use system for general home automation needs. This work was developed from February to July 2017 and hosted by the Onda Technology Institute.

Preliminary research heavily suggested the use of I2C technology for the development of the local cluster networks. However, one of the core design intent requirements of the system was the automatic configuration of devices joining the network, either before system power-on or in a hot-swap fashion. As such, several automatic configuration solutions were studied and considered, including some already known interfaces and protocols, such as SMBus, and their pros and cons weighted against the needs of the system.

The chosen method of configuration was through a combination of a hardware-based token passing system between devices through simple signals, along with a simple and modular two-message protocol approach. This method allowed for a prototype capable of supporting multiple devices with different functions, chosen in the context of a typical home automation device.

In conclusion, this document refers to some unconventional and less known methods to implement automatic configuration in I2C based networks, their advantages and disadvantages, and exemplifies the use of one such method in a home automation oriented prototype.

Resumo

O objetivo do presente documento é relatar a investigação e desenvolvimento efetuados no âmbito do projeto de uma *framework* I2C auto-configurável no contexto de um sistema de automação doméstica, com o objetivo de criar um sistema acessível, seguro e fácil de usar para necessidades gerais de domótica. O projeto foi desenvolvido entre Fevereiro e Julho de 2017, e decorreu no Instituto Onda Technology.

A investigação preliminar efetuada apontou para a utilização de tecnologia I2C para a comunicação com os sensores e atuadores. No entanto, um dos requisitos principais do sistema é a configuração automática de dispositivos que se juntam à rede, quer antes do sistema ser alimentado ou durante o funcionamento normal do sistema (*hot-swap*). Como tal, algumas soluções de configuração automática foram estudadas e consideradas, incluindo protocolos e *interfaces* já desenvolvidas, como o SMBus, e os seus pros e contras contrastados com as necessidades do sistema.

O método escolhido foi uma combinação de passagem de *tokens* entre dispositivos, juntamente com um simples protocolo de duas mensagens. Este método permitiu um protótipo capaz de suportar multiplos dispositivos exemplo com diferentes funções, escolhidas no contexto do típico dispositivo de domótica.

Em conclusão, este documento refere alguns métodos menos conhecidos ou pouco convencionais de implementação de auto-configuração de redes I2C, as suas vantagens e desvantagens, e exemplifica o uso de um desses métodos num protótipo orientado a domótica.

Contents

Abstract **i**

Resumo **iii**

1 Introduction **1**
1.1 Context . . . 1
1.2 Reading Guide . . . 3

2 Problem Characterization **5**
2.1 Overall System Description . . . 5
2.2 Thesis Work . . . 6

3 State of the Art **9**
3.1 Inter-Integrated Circuit (I2C) . . . 9
3.1.1 System Management Bus . . . 10
3.1.2 Space Plug-and-play Architecture . . . 10
3.2 Serial Peripheral Interface (SPI) Bus . . . 11
3.3 IEEE 1451 Standard for a Smart Transducer Interface . . . 11
3.4 Dynamic Host Configuration Protocol . . . 11

4 Proposed Solution **13**
4.1 Solution Overview . . . 13
4.2 Operating Bus . . . 14
4.3 Network Interfaces . . . 14
4.4 Address Resolution . . . 15
4.5 Wired or Wireless Connections . . . 15
4.6 Monitorization Periodicity . . . 15

5 Local Network Configuration **17**
5.1 Network Self Configuration . . . 17
5.2 Discovery . . . 18
5.2.1 I2C Notification . . . 19
5.2.2 Hardware Signal Notification . . . 19
5.3 Enumeration . . . 19
5.3.1 Arbitration based enumeration . . . 19
5.3.2 Device chaining . . . 20
5.4 Configuration . . . 22
5.5 Prototype . . . 23

6 Functional Description and System Components **25**
6.1 Functional description . . . 25
6.1.1 Base Station . . . 25
6.1.2 Nodes . . . 26
6.1.3 Slave devices . . . 26

7 Node and Device Hardware Description **27**
7.1 Level 2 bus . . . 27
7.2 Node . . . 29
7.2.1 I2C Pull-up Resistors . . . 30
7.2.2 I2C Level Shifter . . . 30
7.2.3 Signal Pin Voltage Dividers . . . 31
7.3 Slave Device . . . 31
7.3.1 Temperature Sensor Device . . . 33
7.3.2 LED Controller device . . . 35

8 Control Software **37**
8.1 Software Architecture . . . 38
8.1.1 System Classes . . . 38
8.2 Communication Protocol . . . 38
8.2.1 Device Configuration . . . 39
8.2.2 Regular Operation . . . 41
8.3 Individual device control . . . 42
8.3.1 Nodes . . . 42
8.3.2 Slave Devices . . . 43

9 Conclusions **47**
9.1 Conclusions . . . 47
9.2 Contributions . . . 48
9.3 Future Work . . . 48

A Probability calculation for duplicate tickets **49**

B Analysis of pull-up resistor value versus maximum chain size **51**

C System Classes **53**
C.1 Pin . . . 54
C.2 Bus . . . 54
C.3 Device . . . 54
C.4 Master . . . 55
C.5 SelfSlave . . . 56
C.6 Function . . . 56

References **59**

List of Figures

2.1 High level diagram of the intended system organization. 5
2.2 Intended high-level physical network configuration. 7

4.1 Intended high-level physical network configuration. 14

5.1 I2C interruption example, with Data line interrupted. 21
5.2 Token passing example, indicating the direction of token passing. 22
5.3 Inter-device communication example, assuming a generic two-way protocol. 22

7.1 Connection diagram on slave device. Chain direction is left to right. 28
7.2 Schematic of Raspberry Pi module with associated components. 29
7.3 Prototype Node. 29
7.4 Close up of the signal conditioning board. 30
7.5 I2C transmission over level shifter, with 3.3V side on Channel 1 (bottom) and 5V side on Channel 2 (top). 31
7.6 Base Schematic of the Arduino Nano based slave device. 32
7.7 Schematic of the Arduino Nano based slave device with LM35DT temperature sensor. 33
7.8 Developed prototype for the temperature sensor device. 34
7.9 Close up of the printed board for the temperature sensor device. 34
7.10 Schematic of the Arduino Nano based slave device with the general purpose LED control functionality. 35
7.11 Developed prototype for the LED controller device. 36
7.12 Close up of the printed board for the LED controller device. 36

8.1 Highlight of the local device network in complete system architecture. . . . 37
8.2 System simplified class diagram. 38
8.3 Protocol message format. 39
8.4 Configuration phase overview. 40
8.5 Node State Machine. 42
8.6 Slave Device State Machine. 44

9.1 Developed prototype in chain arrangement; from left to right, Raspberry Pi, signal conditioning board, LED device and temperature sensor device. . . . 48

C.1 System simplified class diagram. 53
C.2 Expanded Pin classes for ATmega328 (left) and Raspberry Pi (right), with common members and methods underlined. 54

C.3 Expanded Bus classes for ATmega328 (left) and Raspberry Pi (right), with common members and methods underlined. 55
C.4 Expanded Device class for the Node. 55
C.5 Expanded Master class for the Node. 56
C.6 Expanded SelfSlave class for the Slave Device. 57
C.7 Expanded Function class for the Temperature Sensor Slave Device. Underlined methods are required by interface. 57

List of Tables

7.1 Connection lines and respective abbreviations. 28

8.1 List of instructions for devices. 39

A.1 Probability of ticket collision as per model A.1 49

B.1 Maximum number of devices for typical I2C configuration values 52

Chapter 1

Introduction

Contents

1.1 Context **1**

1.2 Reading Guide **3**

1.1 Context

Home automation systems are one of the faces of technology as a commodity. The fantasy of having a smart house with robots doing chores, answering to voice commands and essentially being butlers is the main aspect being sold by companies with some mainstream public visibility.

One of the biggest and best known technology companies in the world, Google, has just last year officially entered the home automation field with their solution, Google Home[1]. However, Google Home's product overview [2] features an *applied to the home* version of their typical profiling services, like crossing search, browsing and location data to provide smarter answers to voice commanded questions. Device control is again primarily presented as a voice command option, and only mentioned to be able to control other smart devices, hinted to be limited to devices capable of hosting an Android operating system. Another similar solution is Amazon's Echo device, available since July 2015. Once again, the Echo's offers seem focused on voice controlled applications in the same vein as Google Home: voice controlled Internet searches, updating and consulting appointments, and playing music [3].

Alongside the problem of little commercial visibility, other solution have other problems. Protocols are aplenty, with solutions such as C-Bus [4], Zigbee [5], Insteon [6] and KNX [7], among others; such diversity fragments the market, decreasing overall trust in this type of service, as components are less interchangeable and more expensive to replace. Even for the more widely adopted standards, such as KNX, royalties and licensing [8] increase the

price of home automation, putting even the most modest solutions beyond the capabilities of most demographics.

According to ICONTROL's Smart Home Report 2015 [9], consumers have shown preference towards security and energy efficiency and saving, with entertainment only recently starting to emerge as a driving factor. While these systems promise some of these features, they are tied to already existing smart devices, which are often very expensive. For example, Google Home features light control for Samsung SmartThings lights, which cost upwards from $14.99 per LED light. The report also notes that consumers aren't looking for the most technologically advanced solutions, but rather they look for easy-to-use solutions.

As such, these systems are often prohibitively expensive for a large potential costumer segment, since they require a very large investment on system and peripherals. Solutions that make the most of already owned devices and appliances would make for greater adoption rates.

Another problem is their often isolated development, lacking an integrated ecosystem, meaning that a product of this kind with less commercial success may lead to the discontinuation of the system by the selling company. Abandoning products with such a long expected life cycle means reduced or absent support to the buyers, and in some cases complete shutdown of the project, leading to reduced trust in the company, in the market and in the product as a whole. In one such case[10], Revolv, a small home automation company acquired by Alphabet, Google's parent company, was eventually completely shut down and forced to stop their services, rendering their $300 device, the Revolv Hub, useless.

Finally, as IoT systems, home automation systems are often associated with the concept of wireless communication. While this technology discards cumbersome and expensive cabling, the need for complex standalone systems for each node and the system complexity requirements for hosting wireless communication increase system costs and decrease security. System back ends present in IoT operating systems have aided Distributed Denial of Service attacks[11], and the potential insecurity of these systems not only reduces consumer trust, but inhibits the development of automated systems for more critical purposes, 3such as remote access control to premises.

The Onda Technology Institute, the proponent of this project, believes in a home automation system that surpasses these problems through intelligent and scalable design. Scalable design not only opens the product to a larger market, from middle class homes all the way up to construction companies, but also improves system lifetime and support by allowing third parties to develop and support their own compatible peripherals. We believe this latter point to be key; by creating an open competitive market for third party peripheral development, commercialization and support, the system's overall quality and price will be improved, making for a broader consumer segment that will purchase the system with more confidence.

The main focus of this document is the first phase of this project, with the goal of

designing a platform suitable for hosting a local network of smart devices to serve as the moving parts of the home automation system. This network is overseen by a network master, which will have the main features of configuring the devices and performing the requests for information and actions. The network is intended to be flexible, self-configuring and reliable. It should be noted that although the platform is expected to be generic and applicable in other areas, all design and development will be in the context of home automation systems, and decisions will be made with that in mind.

1.2 Reading Guide

In chapter 2 the main points of this work are detailed, in light of the overall intended system design. Chapter 3 is a State of the Art analysis of relevant systems, concepts and standards. Chapter 4 considers the overall proposed solution to this work, expanding on some of its requirements and choices. Chapter 5 describes considered methods for enabling network self configuration on the I2C bus, along with their characteristics, and the final choice made for this project. Chapter 6 introduces the prototype description as a proof of concept of the planned system. Chapter 7 presents the physical description of the system prototype, schematics and components. Chapter 8 describes the system software architecture, control of devices during typical usage, and description of the adopted internal protocol. This document concludes with chapter 9, along with original contributions and possible future work.

Chapter 2

Problem Characterization

Contents

2.1 Overall System Description **5**

2.2 Thesis Work **6**

2.1 Overall System Description

The generic high-level architecture for the envisioned system, as proposed by the Onda Institute, is composed of three levels, as represented in figure 2.1. While the development of the top level of the system is not the goal of this work, it is important to state the intended features that make up the complete vision of the system, in order to establish the context behind the decisions made.

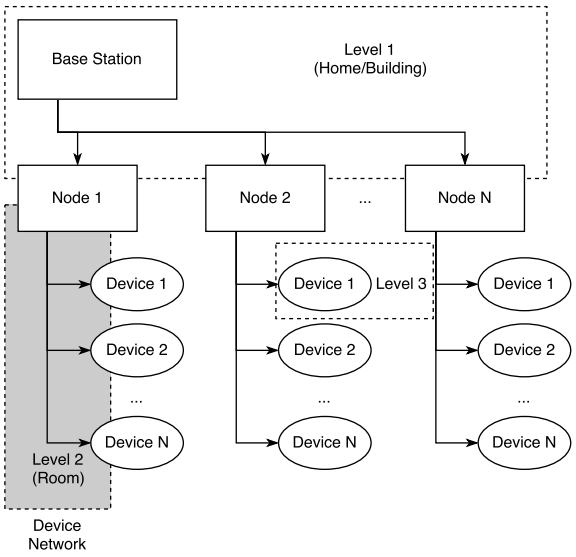

Figure 2.1: High level diagram of the intended system organization.

The highest level is the base station. Its purpose is twofold: it is both the main interaction interface for the user, and it will store all important data required and sampled by the system. User profile based management and task scheduling will be handled by this level, with orders being sent to the device networks. Orders, scheduling and profiles can be made remotely; intended design envisions control through computer program, web interface and smartphone application. By restricting Internet access to this level, the security of the remaining levels can be increased. One base station is intended to control all networks in a building.

The second level is that of the nodes. These nodes are important for end-to-end communication and network configuration, the latter of which is the focus of this work. It is interesting to note that while the base station may be very distant from a node, a device network is designed to cover smaller areas such as rooms, parts of larger rooms or clustered devices. Considering the cabling needs of operating over these distances, there is a possibility of using smaller cables on the local network, and a more expensive long distance cable to connect nodes to the base station; when we consider the larger amount of devices in this level of the system, it is feasible to expect an overall reduction of the cost of implementation.

The third level is that of the devices. Design intent is to have a self configuring network, and to have the option to outsource device development. This requires an easily configurable interface between master and device. The use of common commercial sensors and actuators may not always be possible, either because they lack a controller or because they don't have a configurable one. To reduce the complexity and cost of components, devices should be hosted by micro controllers, a decision that brings some implications in terms of usable technologies.

2.2 Thesis Work

The work on this thesis is the development of the level 2 network, capable of hosting devices based on micro controllers, while respecting the spirit of the planned system, with requirements such as:

- flexibility in hosting devices with various different functions;
- modularity in the interface, to promote third party interest in development of devices;
- budget oriented, namely by using devices based on micro controllers;
- security, namely by using cabled techniques;
- ease of use and device organization comparable to similar wireless devices.

Figure 2.2 presents the planned physical architecture for the bus. This chain architecture allows for the bus to be carried by the devices themselves; using this architecture

instead of a physical holding bus or channel is a reduction in cost, even if an arguably small one, but most importantly doesn't implicitly limit the number of possible devices.

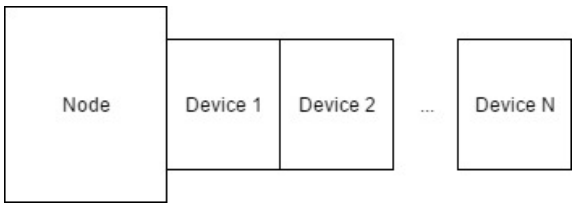

Figure 2.2: Intended high-level physical network configuration.

In the following chapters we address the technologies available to implement a working prototype of this network, along with some others that were studied to inspire new methods and techniques.

Chapter 3

State of the Art

Contents

3.1 Inter-Integrated Circuit (I2C) 9

3.1.1 System Management Bus 10

3.1.2 Space Plug-and-play Architecture 10

3.2 Serial Peripheral Interface (SPI) Bus 11

3.3 IEEE 1451 Standard for a Smart Transducer Interface 11

3.4 Dynamic Host Configuration Protocol 11

In this chapter we provide a view over some of the technologies applicable to the development of the use case project, including I2C-based systems with some capacity of slave configuration, a description of their characteristics, advantages and requirements.

3.1 Inter-Integrated Circuit (I2C)

The Inter-Integrated Circuit (I2C) [12, 13] bus was developed in the 1980's by Phillips Semiconductors, now NXP Semiconductors, as a way to easily establish communication between devices on a single board or card - specifically, to connect a CPU to peripheral chips in a television set. Peripheral devices at the time required too much wiring and additional logic to decode device addresses; Phillips' Eindhoven labs took to creating a bus that would reduce these needs, so as to increase simplicity and reduce costs.

As such, I2C is a two line, half duplex communication bus used to connect integrated circuits over the short distances characteristic to printed circuit boards. Revisions and upgrades to the bus were made to improve functionality, including 10-bit addressing, the 400kbps Fast Mode, the 1Mbps Fast Mode Plus and the 3.4Mbps High Speed Mode.

I2C is nowadays a world standard, being present in thousands of integrated circuits and manufactured and supported by companies worldwide.

In the context of this work, usage of I2C is very obvious, being one of the two most prominent communication options for micro controllers. It has a number of features that can be tuned towards various needs, which promotes flexibility; on the other hand, the lack of a native method of automatic configuration goes against ease of use, which requires consideration.

3.1.1 System Management Bus

The System Management Bus (SMBus) is a derivative of the I2C bus. SMBus specification was defined in 1994 by Duracell International Inc. and Intel Corporation.

System Management Bus presents some differences to simple I2C in hardware standards, such as logic levels, pull-up resistor values and various speed concerns, such as rise and fall times of signal[14].

SMBus does, however, present a native method for address resolution. The network master issues commands to a specific address that is read by all devices. Collisions in the I2C bus are resolved using the I2C arbitration method - digital lows override simultaneous digital highs, and devices verify if their transmission has been suppressed by another device's transmission. Should it be, the device returns to not-addressed slave mode and awaits a new call.

3.1.2 Space Plug-and-play Architecture

The Space Plug-and-Play Architecture (SPA) [15] is a standard designed by the United States Air Force Research Laboratory in an effort to reduce the complexity of satellite systems and in turn their development and construction time. The focus on a standardized interface allowed ease of software development of components, increased robustness of the system, agile accommodation of requirement changes, and increased competition in the component industry.

Among the characteristics of this standard we note the address resolution method, which makes use of a global unique identifier (GUID) and general calls for connected devices. The network temporarily becomes multi-master during the configuration phase, as devices verify addresses until they find one that is free, take said address and finish their configuration procedure. They are then enumerated and identified by the master; this phase includes device self testing, version confirmation and TEDS update[1]. There exists a routine phase where new devices may enter the network, or connected devices may unsubscribe the network.

Further reading evidences other aspects of the standard, such as the use of hubs to interconnect networks; however, we consider these unnecessary in function and impractical economically. Furthermore, the lack of detail in the available literature was an impediment to using this standard.

[1] See section 3.3

3.2 Serial Peripheral Interface (SPI) Bus

The Serial Peripheral Interface Bus was developed in the late 1980s by Motorola as a means to achieve short distance communication between devices in embedded systems. Present in multiple types of applications to this day, it has become a standard in embedded communication.

SPI presents many advantageous characteristics, such as full duplex, serial, synchronous communication between devices, variable message size and slave selection when transmitting [16].

However, SPI is colloquially known as a four-line bus, which raises concerns about the potential clutter of cables. Additionally, connecting multiple slave to a single master often requires additional expansion of its hardware capabilities, notably through port expanders. A technique called *daisy-chaining* [17] presents a simpler solution to this problem; however, this solution heavily relies on all devices functioning properly, since they are used as middlemen by the master to reach more distant slaves. This reliance was the main reason why SPI was abandoned in this project.

3.3 IEEE 1451 Standard for a Smart Transducer Interface

The IEEE 1451 family of standards for Smart Transducer Interfaces [18] covers the addition of plug and play capabilities to analog transducers. It does so by standardizing a Transducer Electronic Data Sheet (TEDS), which contains the data necessary for identification, characterization, interface and calibration of the transducer. TEDS can be present in the embedded memory of the device itself, or as a virtual data sheet that the device can point to and can be downloaded from the Internet.

The standard contains a well defined template for the organization of the in-memory TEDS and for the identification of functions per transducer type [19].

In the context of this work, the usage of the IEEE 1451 Standard is very relevant. The aspect of a downloadable virtual data sheet is very attractive when considering the ease of connection to the Internet through the first level of the system (refer to figure 2.1). The adherence to international standards is also a very good step towards a system with high compatibility.

3.4 Dynamic Host Configuration Protocol

The Dynamic Host Configuration Protocol (DHCP) [20] is a standardized network protocol for Internet Protocol (IP) networks. This protocol is capable of dynamically distributing network configuration parameters, such as addresses, to devices and services in the network. While the protocol itself cannot be used in the scope of this work, since it is designed for Internet Protocol networks, there is merit in studying its design concepts.

DHCP distributes parameters in a discovery-offer-request-acknowledgement (DORA) cycle. In the discovery phase, a new device broadcasts its MAC address using the standard broadcast address 255.255.255.255. The DHCP server then reserves an address for the client and offers said address to the client. The client then requests the offered address. The server then enters the acknowledgement phase, offering the parameters needed for configuration in the negotiated address.

Such a cycle applied to this project would require the auxiliary request line for devices referenced in chapter 2, since discovery is initiated by the client, and a GUID as a default broadcast address, as referenced in section 3.1.2. There are also preliminary concerns with multiple devices joining simultaneously, or a network master initiating with multiple devices connected.

Chapter 4

Proposed Solution

Contents

4.1 Solution Overview 13

4.2 Operating Bus 14

4.3 Network Interfaces 14

4.4 Address Resolution 15

4.5 Wired or Wireless Connections 15

4.6 Monitorization Periodicity 15

This chapter details the proposed solution and expands on some system requirements, how those requirements are met, and the implications of the adopted solutions on the work and system as a whole.

4.1 Solution Overview

The proposed solution is the implementation of the device network based on I2C [13]. I2C has been a very common protocol in commercial electronics. Most I2C systems also use 7-bit addressing, allowing for a total of 128 addresses controlled by a single master, which guarantees reasonable scalability. Finally, the physical bus architecture allows, if needed, for a better extension of the network over any number of devices.

The intended design allows for extension of the network using the devices themselves. Each device will connect to the one before it, and have connectors for the next. This makes for an attractive design, easy expansion, and robustness in the network configuration, as will be explained later. While this design does make for difficult access to devices in the beginning of the chain, this was considered the best implementation overall.

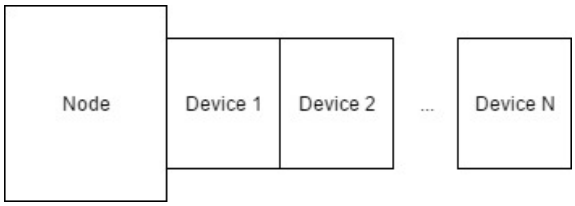

Figure 4.1: Intended high-level physical network configuration.

4.2 Operating Bus

The choice of I2C as the bus for this system level is dependent on multiple factors. A simple and lightweight bus was desirable, both in terms of overall complexity and cabling, to ensure compatibility with as many components as possible and to reduce cabling costs over large numbers of connected devices. I2C presented one such solution, which combined with the overall familiarity with the technology presented a suitable solution.

Notably, one discussed solution was the use of the SPI bus in a daisy-chain configuration. This technique presents some desired characteristics, such as full duplex communication and higher bit rate in transmissions; even though daisy-chaining is still heavier in cabling than I2C, it could present other desirable characteristics. However, SPI daisy-chaining is dependent on devices as middlemen to reach its end point. This design has a glaring flaw in the case of device malfunction, as at least the whole network beyond that device is inactive for as long as that device is malfunctioning. The planned approach is more resilient towards such problems, since requests are made directly to each device.

4.3 Network Interfaces

Usage of the I2C bus presents another difficulty: bare sensors and actuators either have analog interfaces or incompatible digital interfaces, or their I2C programming usually comes preconfigured with a device number or interface identification. It would prove difficult to manage multiple devices if they have fixed identification.

The proposed solution is to make the smart devices with two levels themselves. By using a micro controller unit as a buffer between master and the actual sensor or actuator, many advantages can be gained. The first is a flexible interface with the network, that can be used with any sensor or actuator as long as the MCU can correctly interpret and prepare the data for communication. This allows for a universal protocol between device and master, which is independent of the specifics on the sensor itself, like output type or preconfigured protocols. The second is the guarantee that a capable enough processor is present to take part in the network auto-configuration. This capability is both in terms of performance and flexible programming, since I2C capable MCUs can enter networks with any identity. The third is flexibility in terms of development of devices; given that the requirements for MCUs are intended to be kept modest, there is further space for design

economy on simpler devices, or for added functionality on devices hosting more advanced MCUs.

4.4 Address Resolution

It is notable that I2C has no native support for address resolution. Address resolution being a major part of auto-configuration requires additional procedures to be applied to the system. There has been some research work in this area, with two of the most prominent solutions described as follows.

System Management Bus is an I2C derivative protocol that presents, among other unique characteristics, a native resolution protocol. This is a useful characteristic, as it makes for a standardized solution, already known and implemented.

Another solution would be Space Plug-And-Play Architecture[15] (SPA). SPA address resolution will be explained in chapter 3, but some key points are the usage of a global unique identifier (GUID) for devices which allows them to briefly act as masters and initialize themselves with a unique network identifier.

4.5 Wired or Wireless Connections

As an Internet of Things oriented project, the absence of wireless communication may be questioned, as it is heavily advertised by most already implemented solutions of this type, and it makes for easy distribution of devices in typical domestic space. There were two main reasons why we chose wired instead of wireless for this project. The first reason was security; wireless devices can be easily accessible by outside agents, while a wired network makes for a more resilient system. Somewhat recently, a massive Distributed Denial of Service (DDOS) attack was perpetrated with the help of over 145,000 IoT devices[21], showing how vulnerable wireless based devices are. However, more critical than being part of a DDOS attack, ensuring the devices are suitably protected from outside agents is key. The consequences of outside agents having access to devices that control door access, electricity or gas can result in heavy loss of resources and lives. While there are ways to increase the security of a wireless network, there is a heavy preference from the proponents towards the inherent security properties of a wired network. A second reason was energy efficiency and transportation, as wireless devices require their own power sources, while a wired network can share the same source.

4.6 Monitorization Periodicity

A final aspect of this implementation is saturation of the network capacity. Monitoring devices are typically managed in a periodic network sweep. However, the large amount of devices of this type that could potentially be installed in a single network presents a

challenging worst case scenario when it comes to bus saturation. Using event based communication, for example when a sensor measures a minimum difference in values over a certain amount of time, is challenging for a different reason: I2C has no native support for slave requests. Using an auxiliary line, the device's MCU can represent a slave request, prompting an I2C request from the master. The master then broadcasts to all devices in search of the request origin. Using a reserved I2C address for broadcasting, in addition to the dynamic network address, would be the implementation with best theoretical response time, but it quickly presents probabilistic problems when simultaneous requests are considered. Alternatively, the auxiliary line could be used as a trigger for the network sweep, offering the same functionality as a periodic sweep but potentially saving computational power on the node on lower activity periods.

Preliminary research shows that broadcast is also not native to some microcontroller libraries. As such, an individual network sweep is necessary. This method implies a potential long response time, should the requesting device be very distant from the master, but this can be alleviated with a recommendation to install more critical devices (for example a smoke alarm) closer to the master.

Another discussed solution is to use an additional parallel communication protocol, such as 1- Wire[22] or a similar lightweight bus, to send the device identifier along with the slave request. While this implementation may be better for networks with many devices, testing is necessary to show speed differences between the two. There are also worries with additional cabling and compatibility issues. Additionally, the necessary libraries to be installed in the device MCU to support this implementation may be so storage memory intensive that it will negatively impact the programming necessary to operate the sensor or actuator in the device.

Chapter 5

Local Network Configuration

Contents

5.1 Network Self Configuration 17
5.2 Discovery 18
5.2.1 I2C Notification 19
5.2.2 Hardware Signal Notification 19
5.3 Enumeration 19
5.3.1 Arbitration based enumeration 19
5.3.2 Device chaining 20
5.4 Configuration 22
5.5 Prototype 23

In this chapter we will address the design and development of the self-configuring aspect of the local device networks.

As stated in the project introduction, the design intent of the system is one where the system provides a baseline for easy installation of home automation devices. These devices, although varying in function, should have a common interface with the node, to facilitate organization and configuration of the devices. As such, all considered designs are approached under the use case of the project system prototype.

5.1 Network Self Configuration

A self configuring system is commonly known as a system where devices need very little to no user input to be functional in the system they are installed in. Besides tasks like physically handling device installation and other physical interactions, the goal of this project is to require no more from the user than to update their own profile on their preferred device. For example, while a temperature sensor may natively warn the user of

extreme heat or cold, the user may want to set one or more thresholds where the system may automatically regulate other devices, like air conditioners.

Some terminology was adopted from other studied technologies to describe the phases of the self configuration process.Three main phases are considered:

- **Discovery** of the devices refers to the ability of both device and node to recognize the device's presence in the network. The term discovery was adopted from the Dynamic Host Configuration Protocol (DHCP) phase of the same name. In the protocol, it relates to client and server recognizing each other, and is the first step of configuration. However, its features can't be directly translated to a simpler protocol like I2C. For one, as an application layer protocol, it handles devices with far more capabilities than a MCU, such as storage and application memory. Additionally, DHCP operates with client-triggered discovery, which in I2C could imply a multi-master network, if only temporary, which introduces the chance for collision of transmissions. Single master I2C buses, however, have little to no chance of collision, so it becomes important to weigh both factors when making a decision.

- **Enumeration** or **addressing** refers to the process in which the node organizes the devices as generic network slaves or clients, that is, they are given addresses before their actual function becomes relevant. One of the most important phases of this process, enumeration is particularly challenging because simple I2C is designed to work with fixed, previously known addresses. While some MCU interfaces are quite capable of changing operating addresses, there is the issue of distinguishing them before that point, so they can be organized. The SMBus variant interface uses its own address resolution protocol, but implies other design requirements, such as reduced bus speed; other software-based implementations could have similar impact. Hardware solutions could be inelegant, but can be made to not be very intrusive on the standard I2C functions.

- **Configuration** refers to the phase where the devices are functionally identified; there is a critical process of identifying which functional requests the node can make. In this project, the configuration step refers to the identification of the device's functional characteristics, such as TEDs. This step goes beyond the scope of the local network, as such information would likely be introduced through an Internet connection, and the base station is not the scope of this project. Still, aspects of this step, such as device identification through serial number or code, will be implemented in advance, to serve as basis for further development.

5.2 Discovery

The discovery process in an I2C network is straightforward for the slave device, since it can easily be designed in a way that it can consider itself in a network for as long as it

has power. The node, however, requires a notification that a new device has joined the network.

5.2.1 I2C Notification

This method of notification is a simple I2C transmission from the smart devices. This requires slave devices to temporarily take the role of masters, and uses typical arbitration. Addressing is specially important with this technique: the node must have a unique predefined address, and it must be the lowest operating address, otherwise it will win arbitration against one or more unconfigured devices, leaving them so for an indeterminate period, potentially for as long as the network is powered. There is also the need to distinguish slave devices, since the joining of multiple slave devices implies multiple I2C masters starting transmissions from the same address. This can be done through arbitration, if devices have unique identifiers such as a serial identification number, and said identifiers are transmitted during configuration. This does however imply a temporary multi-master network, which may not be desirable.

5.2.2 Hardware Signal Notification

This technique uses a simple hardware signal to notify the node that a new device has joined the network. This method can use a shared line for all devices or task adjacent devices to notify the node. While it uses an additional line, it is much simpler to implement and functional in a single master network.

5.3 Enumeration

5.3.1 Arbitration based enumeration

Using the arbitration method native to I2C is a natural option for the necessary address resolution. However, while the characteristics of the arbitration process could be useful in auto configuration, direct adoption of this process is insufficient, since it requires all devices to have unique and preset addresses.

5.3.1.1 Arbitration based enumeration with random number generation

Suppose a single-master local network of M devices. All devices have the same, preestablished broadcast address. At system start, all devices enter the bus with the broadcast address and generate a random N-bit number, which we will call a ticket. When called by the broadcast address, all devices start transmitting their ticket; however, through I2C arbitration, only the device with the lowest ticket will transmit its whole ticket number. As devices back off, they ignore all bus communication through the most appropriate means, until the master is ready to configure another device. This process repeats until all devices

are configured. There is an undesired possibility of duplicate tickets, which can cause a very problematic situation. A detail on the probability of duplication can be seen in annex A.

In the case duplicate tickets are generated, the devices that generated them will transmit them at the same time in response to the broadcast call. Since the tickets are duplicate, neither device will know it is transmitting at the same time as another; the node will not detect this duplicate transmission either. Without further error checking and prevention measures, it is entirely possible for vastly different devices to be leased the same address, which could have unpredictable consequences.

There is a number of other reasons why this technique was avoided for this work. The first was its probabilistic nature, specially if taking into account there are simpler deterministic options. The second was the non-randomness of the generators in the device, since they are seed based; seeds based on powered time are untrustworthy for the scenario where multiple devices are present at the time of system being powered, and using noise as a seed is not a safe option, since the expected proximity of devices implies that any electromagnetic influence on a single device would be mirrored on other devices.

5.3.1.2 Arbitration based enumeration with unique identifiers

Another way arbitration can be used is by having all competing devices temporarily act as masters, and enter a unique predefined identifier, such as a serial identification number. This way, no additional hardware is required, and devices will be configured in ascending order of their identifiers. On the other hand, unique identifiers become much more critical for normal functionality, and as such more coordination is required between third party developers.

5.3.1.3 System Management Bus

The System Management Bus[23] (SMBus) interface hosts its own address resolution protocol and otherwise is functionally very similar to I2C; as such, it was considered for the purposes of this project. However, SMBus operates in the 10k-100kHz range of clock speeds, which is less versatile than I2C which operates at a much larger range, from SMBus speed up to 3.6MHz. Bus speed does scale inversely to the number of devices on the bus due to pull-up resistor requirements, as will be described in further chapters, and for the greater sized networks this speed restriction becomes less relevant; however, this project considers flexibility in network assembly to have higher priority, for example with allowing for faster buses with fewer devices, which would be hampered with the usage of SMBus.

5.3.2 Device chaining

One way to achieve enumeration is to physically organize the devices in the order they are addressed, e.g. the first device has address 1, the second address 2, and so on. This

simplifies the resolution of conflicts, since no software is needed to handle conflicts or collisions, and the order of the devices is often obvious and intuitive even to the user. This design does, however, introduce an additional, if often small, level of hardware.

In the following examples, consider node and devices in a chain network. Each device will have two dedicated configuration channels, C1 and C2. In this, C1 will receive input from the previous member of the chain, while C2 will output to the next. The node will only have C2. These general designations refer to channels that vary from signals to buses, and their presence, absence and details will be described in each case.

5.3.2.1 I2C interruption

In this design, each device only has a dedicated C2 pin, which carries a digital signal to toggle a switch (e.g. a transistor) that bridges the I2C data line (SDA) and the corresponding pin in the next device. As such, a device will only be able to read the I2C bus as long as the previous device maintains C2 active. This activation happens when a device is properly configured and, as such, forces the devices to configure one-by-one, even while using a unique address.

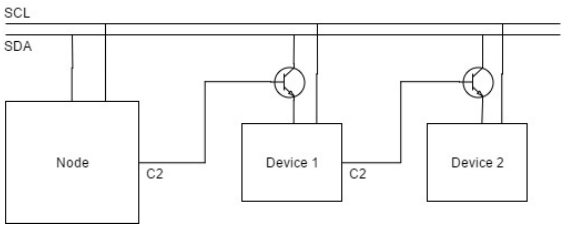

Figure 5.1: I2C interruption example, with Data line interrupted.

While this design requires the least number of effective pins, a device malfunction on any device would disable the following device. Device malfunction during configuration will completely stop configuration of further devices, since they require the previous member to begin their own configuration; malfunctions after configuration have their impact reduced to the following device, which will have its I2C line interrupted, but will still be able to allow communication to the following device. There are also other problems, such as signal distortion by the switch, added capacitance to the line - which reduces the maximum number of devices - and the cost of reducing both effects.

5.3.2.2 Token passing

In this design, C1 and C2 are digital signals used in the MCU's program to make decisions about when to prepare for configuration. When visualized as a network, this would look like token passing - a device receives the token, configures, then passes the token to the next device.

This design is a variant of the previous one, trading an additional pin for the switch, but interfacing directly with the MCU makes it so that the device is independent as soon as it

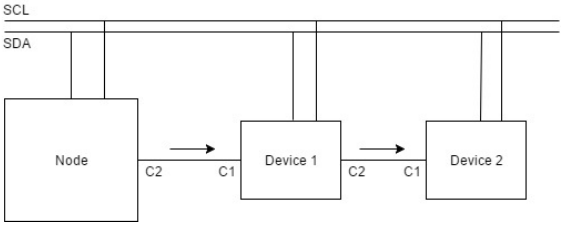

Figure 5.2: Token passing example, indicating the direction of token passing.

is configured; malfunctions will no longer affect the rest of the network beyond preventing the acceptance of a new device that connects directly to the faulty device. This design considers the network a chain only during configuration, while allowing it to perform like a regular I2C network after configuration. The design is also relatively light on additional hardware, and this made it a suitable choice for the enumeration system.

5.3.2.3 Inter-device communication

A discussed design was having C1 and C2 be a communication bus, allowing two sequential devices to communicate addresses - for example, device with address 5 could communicate to the next that it would have address 6. This could shorten the communication cycle with the master during address resolution.

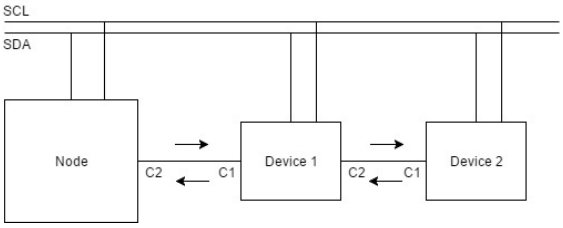

Figure 5.3: Inter-device communication example, assuming a generic two-way protocol.

However, this solution is overly complex for this problem. The implementation of another protocol, along with the potential of requiring many additional pins, depending on the chosen protocol, could increase MCU complexity, which was deemed undesirable. It presented an opportunity for additional functions, but since all were achievable through the means already available, the design was not adopted.

5.4 Configuration

As mentioned before, the configuration step prepares the information the node requires to correctly operate connected devices. This includes information of instruction codes, expected size of messages, data processing and handling of complex instructions.

The IEEE1451 Standard for Smart Transducers presents a standard for Transducer Electronic Data Sheets, or TEDS, to enable plug and play capabilities to transducers connected to microprocessors, instrumentation systems and field networks. TEDS carry information necessary for identification, characterization, interfacing and processing of transducers and transducer data. This information can be carried by the transducer itself, or be accessible from outside the network through a referral to the appropriate file.

The concept was adapted for this system. Devices either carry this information in the associate MCU, or their identification information allows the node to request data sheets from the base station for connected devices as these are configured. These are files hosted on a service-related database, where files can be easily edited by device creators to provide continuous support for devices. Identification information exists in the smart device in the form of a referral link to the database, or the device's model and manufacturer information for an automated search.

5.5 Prototype

For this project, the choice of discovery and enumeration methods were critical, and took many technical and non-technical factors into account.

The chosen discovery method was Hardware Signal Notification, due to the desire to have a single-master network.

The chosen enumeration method was the Token Passing in a Device Chain architecture. Once again, this was due to the intention of having a single-master network, but also to increase system reliability, opting for a deterministic rather than a probabilistic method, and reducing the network interface complexity to allow maximum leeway to third party developers.

Chapter 6

Functional Description and System Components

Contents

6.1 Functional description 25

6.1.1 Base Station 25

6.1.2 Nodes 26

6.1.3 Slave devices 26

As described in chapter 2, the design of the system has three levels: the first level handling human interface and control of tasks through scheduling, environmental triggers and user input; the second level organization and configuration of local device networks; and the third level control of devices, either by gathering and reporting data or performing tasks.

6.1 Functional description

6.1.1 Base Station

The system base station has two main functions. The first is to interface with the user. This is done with an appropriate application, where the user can schedule alarms or periodic tasks, or control connected devices in real time.

The second is to process the data from the devices. This is naturally highly dependent on the type of devices installed, but it can range from receiving environment data such as temperature and humidity, storing said data for statistics, using the data for automated control of related appliances, and checking against typical or user-set limits.

This level is intended to be operated by a machine of specifications comparable to a personal computer, both to relieve the lower levels of computational power and as such

reduce their cost, and to be able to coordinate multiple networks, which includes identifying installed devices and preparing the necessary libraries.

6.1.2 Nodes

The node performs configuration of the devices and is a bridge for communication between both ends.

The existence of this level is intimately connected to the cabling needs of the system. The I2C bus has very short operating range, which makes connecting sensors and actuators, distributed over multiple rooms, directly to the base station difficult. Alternatively, a longer range bus could be used to connect devices directly to the base station, but such a solution would both be bulky and quickly become costly. The use of a middleman device to transition between the two buses allows for better aggregation of devices and a more efficient cabling while accounting for the range needs of the system. The use of Ethernet over small caliber twisted pair cables in lesser instances for the first level along with an I2C bus to connect the more numerous devices is a serviceable and cheap solution. Additionally, allowing the node to take the role of device configuration and organization, in addition to interfacing both buses, reduces complexity of the base station level.

The node is built on the Raspberry Pi 3. This choice was due to availability and similarity to the type of device expected to host the node, as a device capable of both I2C and Ethernet communication.

6.1.3 Slave devices

Slave devices provide the functionality of the system. While functionality will vary from device to device, a common interface is required to ensure the required modularity and seamless integration. The common interface also provides a good base for third-parties to work from.

Prototype slave devices are built on the Arduino Nano board based on ATmega328. This choice of controller was based on board readiness for prototyping, availability and prior experience with the system.

It is notable that ATmega328 officially supports the Two Wire Interface (TWI), rather than I2C. Documentation for the board supports full Two Wire Interface compatibility with I2C at normal and fast speeds, and in this document they were considered equal unless noted.

One of the slave devices hosts a LM35DT temperature sensor[24]. This sensor was chosen due to availability, since the specific capabilities of the sensor are not the focus of its usage; rather, it serves as a representative of the typical sensor equipped slave device for the purpose of testing expected system behavior.

Another slave device hosts three general purpose LEDs to serve as a representative of typical actuator equipped slave device for the purpose of testing expected system behavior.

Chapter 7

Node and Device Hardware Description

Contents

7.1 Level 2 bus 27

7.2 Node 29

7.2.1 I2C Pull-up Resistors 30

7.2.2 I2C Level Shifter 30

7.2.3 Signal Pin Voltage Dividers 31

7.3 Slave Device 31

7.3.1 Temperature Sensor Device 33

7.3.2 LED Controller device 35

In this chapter the hardware description of the system, such as system schematics, development of the physical component of the system and choices in components is presented.

As the system is modular, the Level 2 bus[1] will be described first, and bus lines referred to by function, establishing them before their usage is detailed.

7.1 Level 2 bus

The Level 2 bus includes the necessary functional lines of the system as well as power carrying lines.

The functional connection is a five line bus, mainly divided between the I2C data and clock lines; and the three digital signal lines, that carry the interrupt line, the configuration token, and presence data.

[1] Refer to figure 2.1.

Table 7.1 features the described lines and their abbreviations as used in system schematics further in this document. Figure 7.1 depicts the connection of both interfaces in a single device, as described above.

Line	Abbreviation
I2C Data	SDA
I2C Clock	SCL
Interrupt	INT
Configuration Token (receiving)	CONF1
Configuration Token (passing)	CONF2
Presence Signal	PRSC1
Presence Sensor	PRSC2

Table 7.1: Connection lines and respective abbreviations.

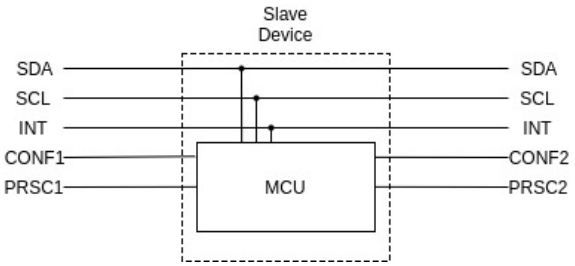

Figure 7.1: Connection diagram on slave device. Chain direction is left to right.

Each device, with the exception of the node, possesses two interfaces for this connection, one for interfacing with the previous member of the chain, and one for the following member. The node, as the mandatory first member of the chain, has no interface for a previous device. The I2C data and clock line of both interfaces are directly connected, and the interrupt line is shared by all devices. On the other hand, the configuration and presence lines are not shared, instead allowing adjacent devices to monitor important signals.

The configuration line is critical to device addressing, and as a token passing line, requires direct control from the interfacing device. The arriving configuration line, that is, from the previous member of the chain, is how the device receives the token, while the departing configuration line, that is, the line that carries the token to the following device in the chain, is how the device passes the token.

The presence line is used to monitor the state of devices in the network. Devices are programmed to inform the node should they detect the connection or disconnection of the next device. This is required of a hot-swap system, but also a valuable addition, since should a device in the middle of the chain malfunction without affecting following devices, the node can verify this and stop operating only the faulty device. As such, this line sends a continuous presence signal to the previous device in the chain, while detecting the presence of the following device.

7.2 Node

The node device is hosted on a Raspberry Pi 3. The schematic is depicted in figure 7.5. The Raspberry Pi 3 and the printed circuit board that holds the necessary components for signal conditioning can be seen in figure 7.3; a close up of the printed board can be seen in figure 7.4.

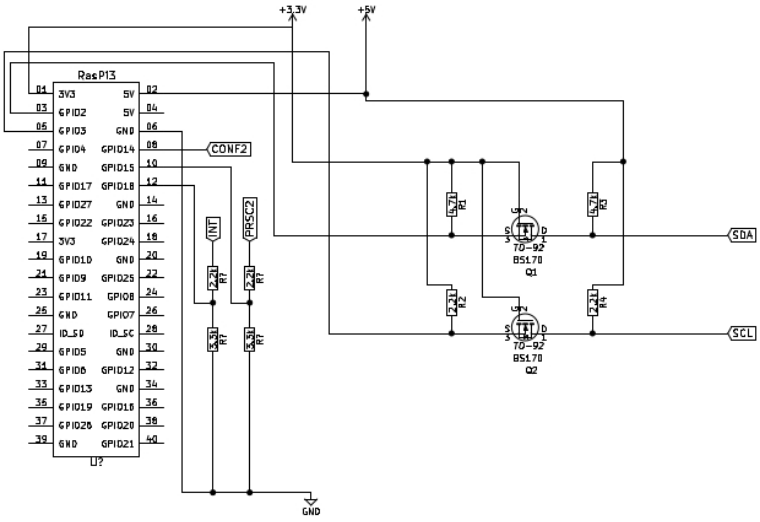

Figure 7.2: Schematic of Raspberry Pi module with associated components.

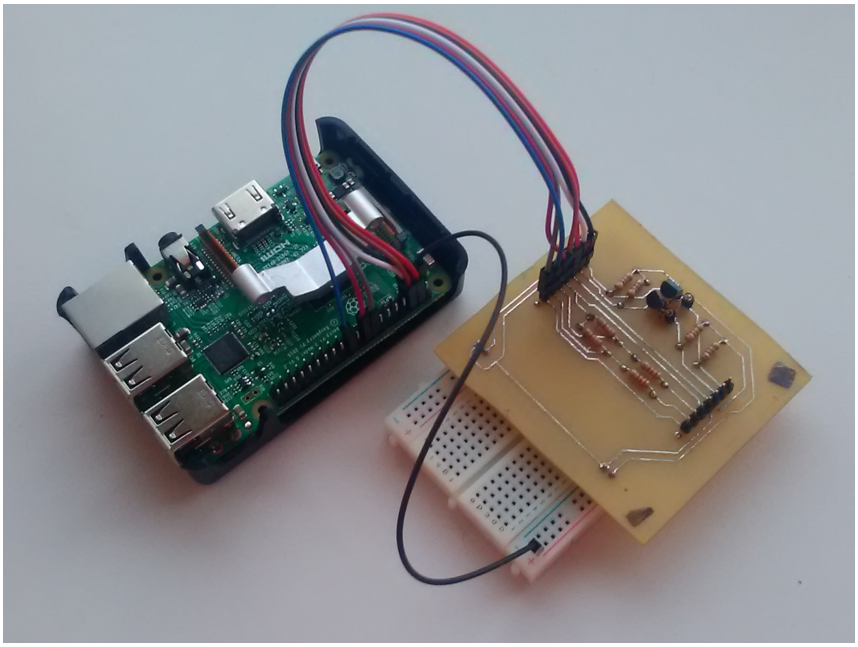

Figure 7.3: Prototype Node.

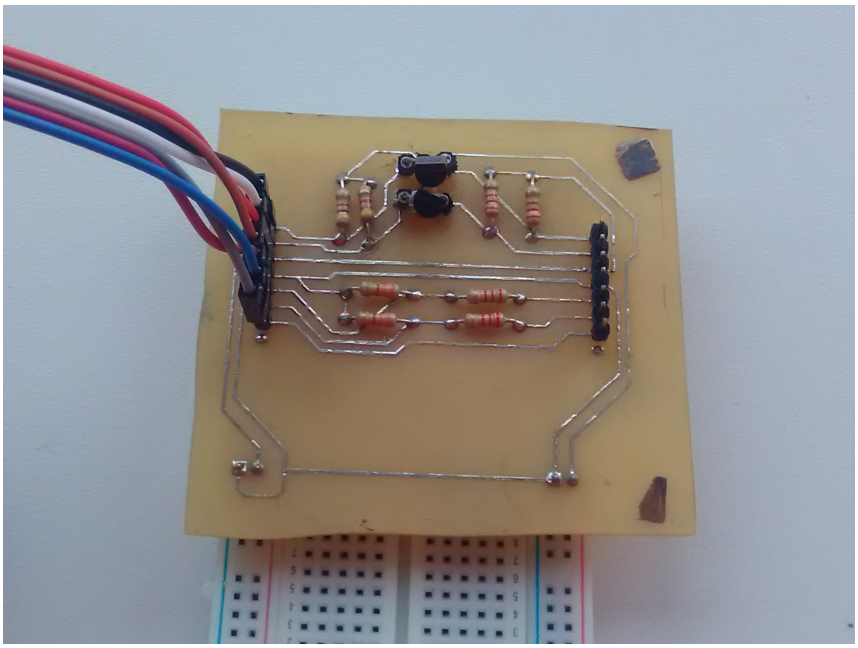

Figure 7.4: Close up of the signal conditioning board.

7.2.1 I2C Pull-up Resistors

The choice of pull-up resistor for the I2C bus is relatively important. I2C specification[12], page 55, states that maximum resistor size scales inversely with bus capacitance. This is to ensure correct signal rise time. However, this means that sizing for the top end of bus capacitance values requires very modest resistor sizes, which simplifies the task of sizing the bus for a very significant amount of devices.

As such, the bus is pulled up by 2.2k ohm resistors on the 5 volt side of the bus, and by 4.7k ohm resistors on the 3.3 volt side of the bus.

A more detailed analysis on the relation between pull-up size, bus capacitance and number of devices can be found in annex B.

7.2.2 I2C Level Shifter

The main point of complexity in the assembly is the I2C level shifter. The level shifter is necessary due to the different operating voltages between the host systems. This is due to the operating voltage of 3.3 volt in the Raspberry Pi, while the output voltage of the Arduino Nano digital pins is 5 volt.

The assembly uses a BS170 MOSFET[25] for each I2C line, allowing the line to simultaneously and bi-directionally support two different pull-up voltages. Since there is only one device requiring level shifting, only one shifter is used.

According to the BS170 data sheet[25], capacitance with small signals such as the I2C communication could be as high as 60 pF, approximately six times higher than the capacitance added by a single ATmega328 based slave device. Since the maximum pull-up

resistor resistance scales inversely with bus capacitance, as per the I2C Specification[12], effectively limiting the number of devices by their added capacitance to the bus, adding too many level shifters would severely limit the number of devices on the bus, which goes against design intent. Considering these capacitance values to be typical among devices of either type, there is a conscious effort to homogenize operating voltage among devices and, in the scenario where such is not possible, to organize devices such that only the minimum amount of level shifters is used.

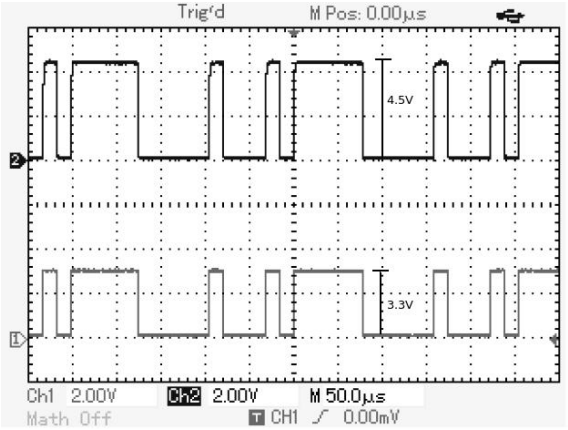

Figure 7.5: I2C transmission over level shifter, with 3.3V side on Channel 1 (bottom) and 5V side on Channel 2 (top).

7.2.3 Signal Pin Voltage Dividers

The Raspberry Pi makes use of passive voltage dividers that exist in the input pins of the Level 2 bus, namely the presence sensor and interrupt lines. The voltage divider uses 2.2k and 3.3k ohm, for a 60% division of the entry voltage, making a stable voltage for simple signal transmission.

7.3 Slave Device

The network slave devices are hosted on Arduino Nano boards with the ATmega328 controller. The schematic is depicted in figure 7.6.

As described before, the base slave device schematic features the interface for device chain integration, keeping the integrity of the I2C bus and all other signals. The simplicity of the board allows for fairly expanded device functionality.

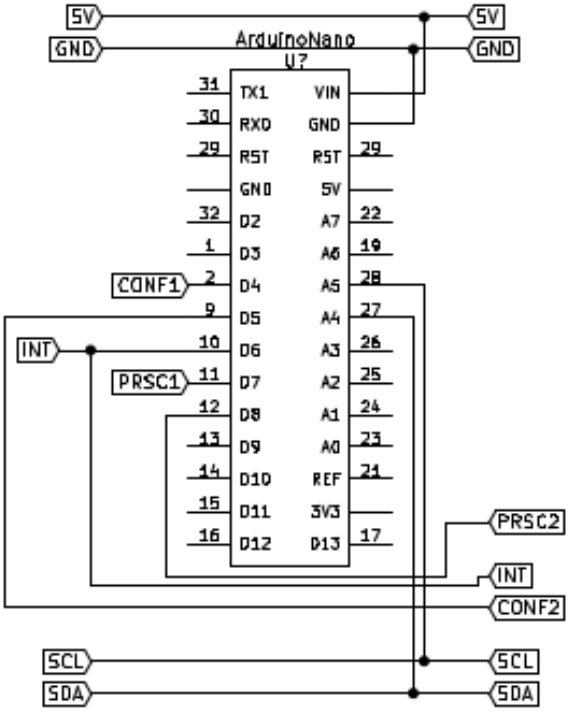

Figure 7.6: Base Schematic of the Arduino Nano based slave device.

7.3.1 Temperature Sensor Device

The temperature sensor device expands on the base assembly with a LM35DT Temperature Sensor, as depicted in figure 7.7. The developed prototype of this sensor is presented in figure 7.11, with a close up of the printed board in figure 7.12.

The LM35 Temperature sensor is in Basic Centigrade Temperature Sensor mode[24] for the sake of simplicity of demonstration of an example sensor-type slave device.

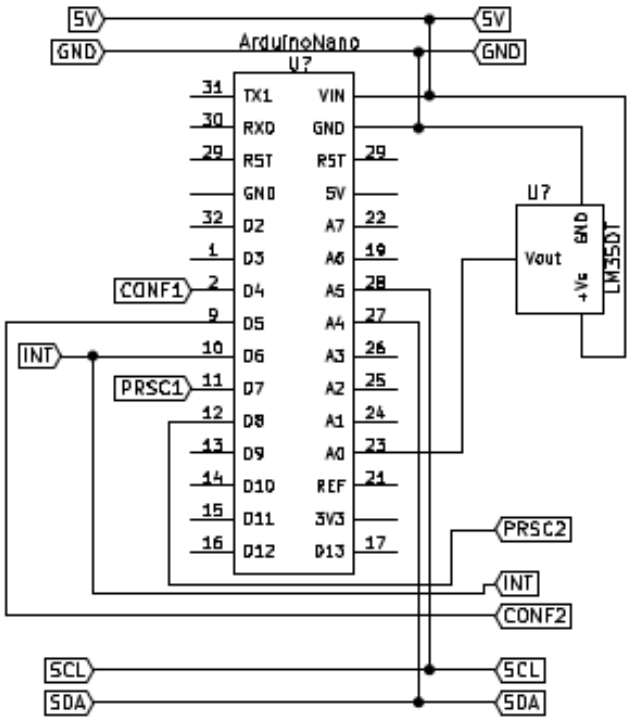

Figure 7.7: Schematic of the Arduino Nano based slave device with LM35DT temperature sensor.

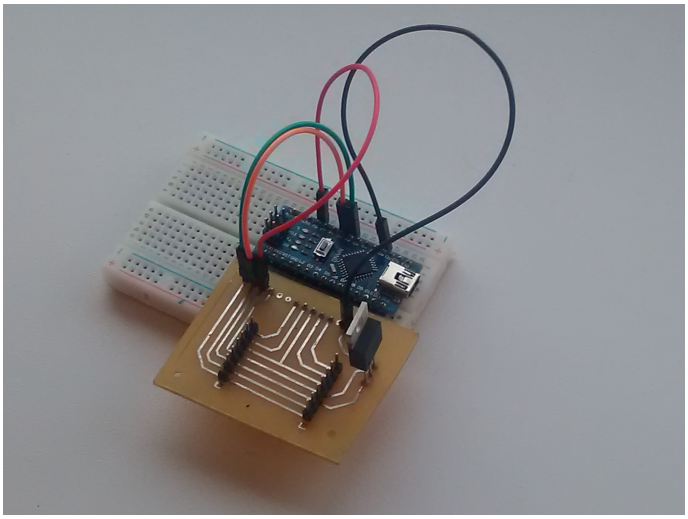

Figure 7.8: Developed prototype for the temperature sensor device.

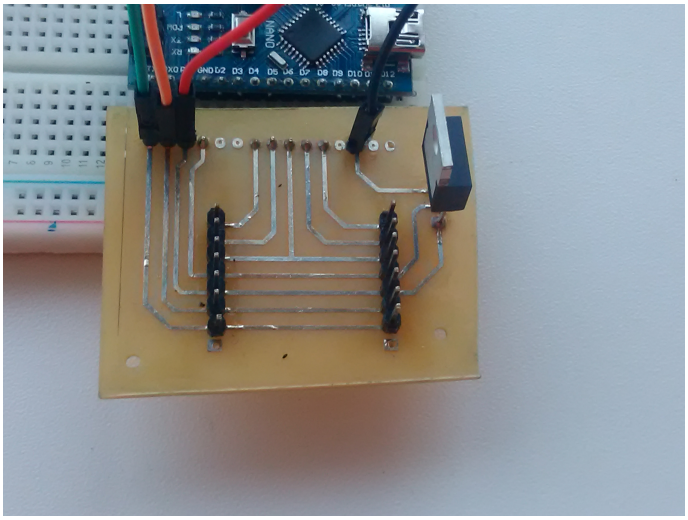

Figure 7.9: Close up of the printed board for the temperature sensor device.

7.3.2 LED Controller device

The LED controller device expands on the base assembly with three general purpose LEDs, as depicted in figure 7.10. The developed prototype of this actuator is presented in figure **??**, with a close up of the printed board in figure **??**. The LED function in this slave device is used as a demonstration of an example actuator-type slave device.

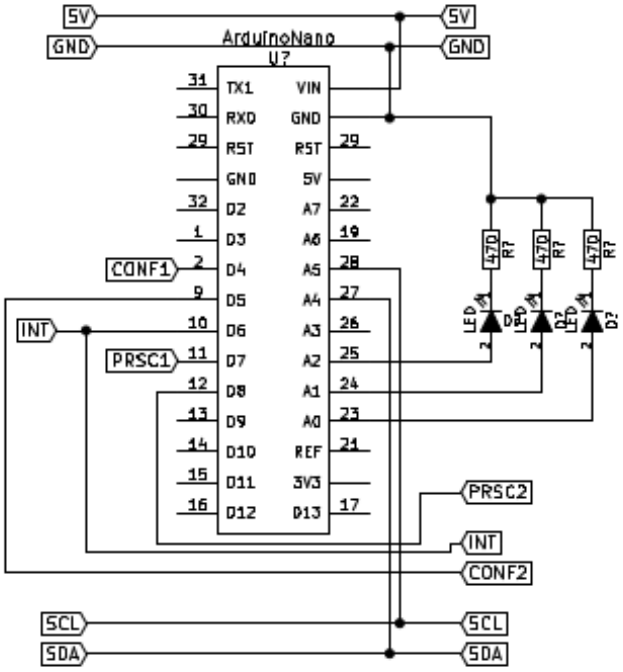

Figure 7.10: Schematic of the Arduino Nano based slave device with the general purpose LED control functionality.

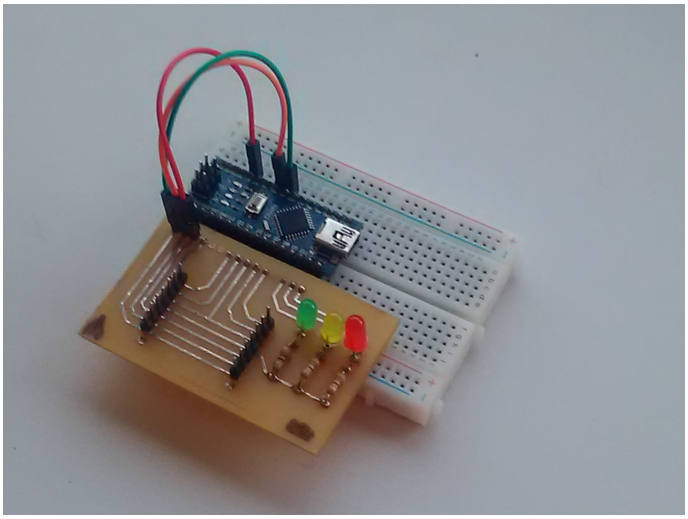

Figure 7.11: Developed prototype for the LED controller device.

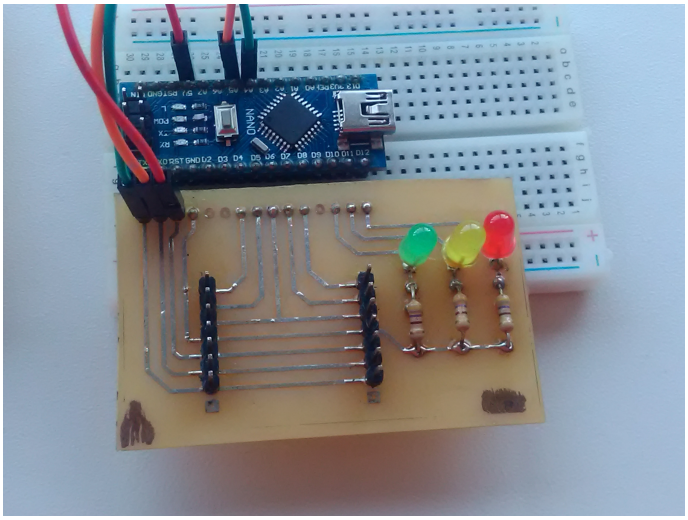

Figure 7.12: Close up of the printed board for the LED controller device.

Chapter 8

Control Software

Contents

8.1 Software Architecture 38
8.1.1 System Classes 38
8.2 Communication Protocol 38
8.2.1 Device Configuration 39
8.2.2 Regular Operation 41
8.3 Individual device control 42
8.3.1 Nodes 42
8.3.2 Slave Devices 43

This chapter describes software architecture, the overarching control scheme, algorithms and state transitions in the control of both interfaces of the local device network, as highlighted in figure 8.1.

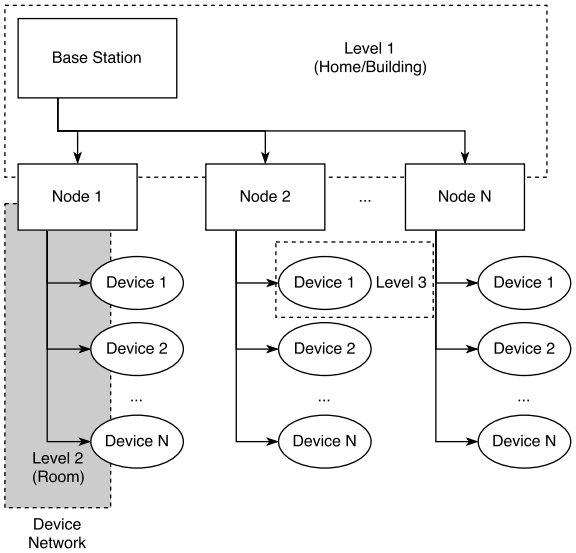

Figure 8.1: Highlight of the local device network in complete system architecture.

8.1 Software Architecture

Software architecture is a critical aspect of a modular system such as this. To ensure costumer safety and satisfaction while being able to accommodate third-party contributions to the system, there needs to be both a tight control on base operations, such as configuration parameters, low and mid level communication compatibility, and overall system security and protection. At the same time, the system needs to allow third parties to access the larger part of the device's capabilities, and require an accessible, powerful and flexible programming interface.

8.1.1 System Classes

The system classes are as depicted in figure C.1.

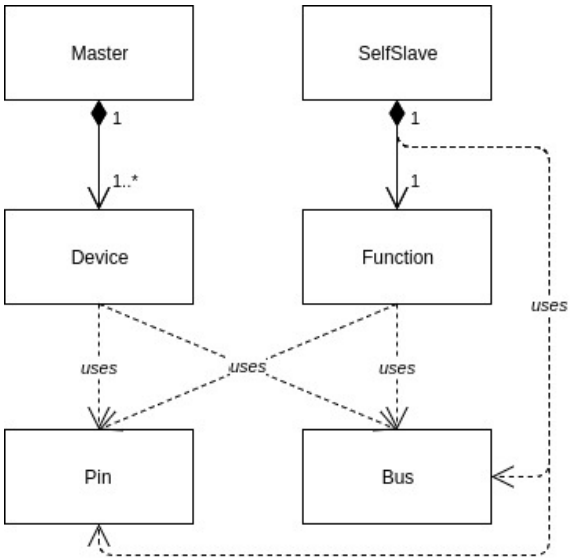

Figure 8.2: System simplified class diagram.

While node and slaves share some high level aspects, lower level characteristics are mainly dependent on platform or system; in this case, those differences are more prominent on how systems handle I2C communication, and how they interface with their input-output pins.

A more detailed description of system classes can be found in appendix C.

8.2 Communication Protocol

The system's communication protocol organizes communication through two byte *instructions*, used to organize reading and writing between device and node. Instructions can be seen as a header for the actual data being transmitted - however, they are effectively an independent two byte message, followed by a variable sized data message.

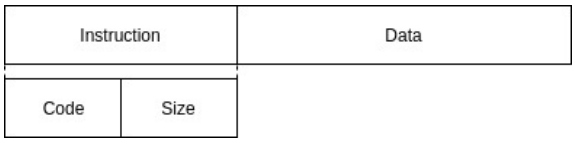

Figure 8.3: Protocol message format.

The instruction transmission is always a Master Write - Slave Read transmission, while the data transmission may be in either direction. Simple sensor devices may find this header unnecessary, but for complex devices with multiple functions, assuring that both node and device know the purpose and size of the transmissions is critical for correct network operation.

The format for instructions is a one byte *code*, followed by a one byte *size of data*. Codes in used as of the writing of this document are present in table 8.1.

Code (Hex)	Description
0x01	Check if exists. Typically for devices in broadcast address.
0x02	Request device manufacturer identification.
0x03	Request device model identification.
0x04	Request device serial number identification.
0x05	Address leasing.
0x06	Request address confirmation
0x07	Return to default address for unconfigured devices.
0x08	Check if device has readings. Used on slave requested network sweep.
0x09	Confirm successful end of configuration.

Table 8.1: List of instructions for devices.

8.2.1 Device Configuration

The configuration phase occurs only once, as the node is being powered on. During this time, a routine where the node prods the network for not-configured devices is repeated for a set number of times. This timeout counter is necessary since there is a chance of an interval between the configuration of sequential devices where no device will react to a node call - for example, while a newly configured device is self-checking and/or changing address, but has not yet passed configuration permission to the next device. This timeout is reset after every successful configuration, so the node must receive no response from the broadcast address for an arbitrary number of consecutive attempts to leave this loop.

During the configuration routine, several data transmissions ensure not only the identification of both devices, but the stability of the connection as well. The data exchange between both device and node is depicted in figure 8.4.

The protocol transmissions, as shown in the figure, take into account the fact that all I2C transmissions must be started by the node.

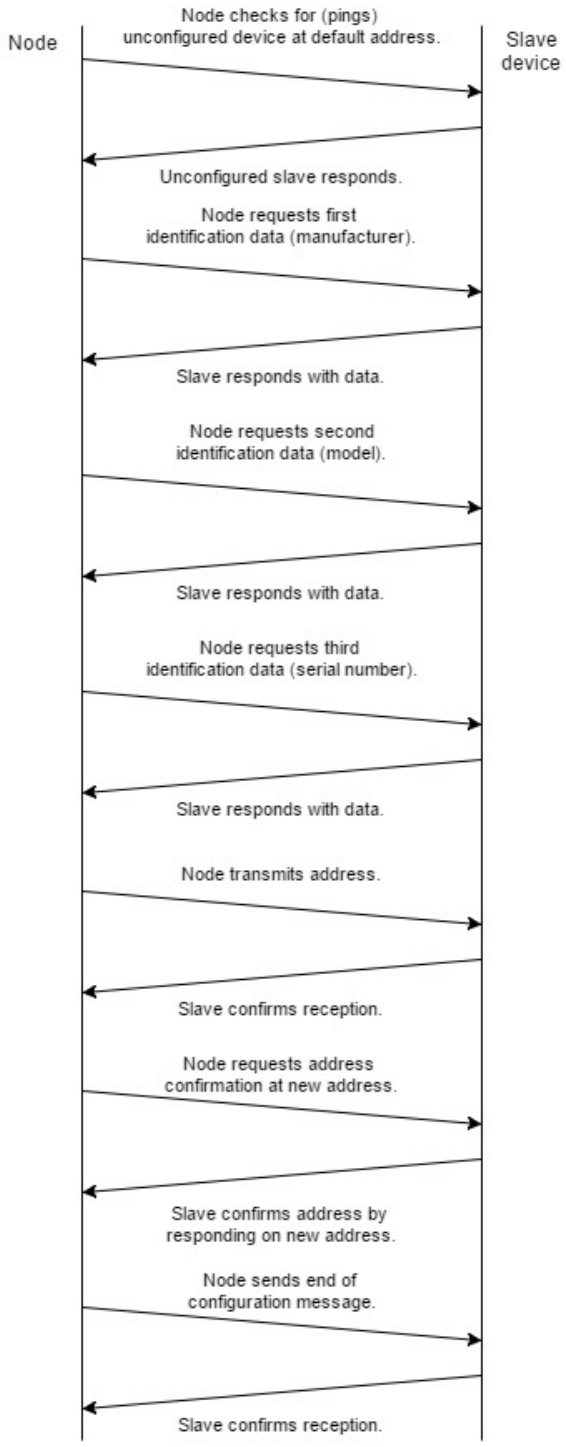

Figure 8.4: Configuration phase overview.

The three identification parameters from the slave device - manufacturer, model, and serial number - are assumed known and registered from a database outside the system. These aren't double-checked during configuration for two reasons. Firstly, the length of these identifiers makes the bulk of the transmission's size, with manufacturer and model

being four bytes in size, and serial number eight bytes in size, such that adding confirmation would almost double the required time. Secondly, identification of the attached devices is both largely irrelevant and easily correctable for the node, while mis-addressing is a much larger concern. Confirmation of the association of identification parameters is performed at base station level, and a repeat request for this information and subsequent correction can be made before any potentially dangerous requests are made from the attached device. Mis-addressing, on the other hand, can both put network organization into disarray and make one or more devices recipient of instructions they either cannot execute, or can execute with some danger, such as tasks designed to be overseen by the user.

The final phase of the configuration process handles the actual address change in the slave device address. Notably, between the lease and confirmation steps, the slave device takes the newly leased address, and the address confirmation step is made on the new address.

8.2.2 Regular Operation

After the configuration phase, the network enters regular operation. During this time, the bus is released by all devices. Transmissions most typically happen in three cases: the node calls a slave device for a scheduled task, the base station level requests a slave device call triggered by user input, or a slave device activates the General Interrupt line.

In the first two cases, both device address and function are known, so the node makes the appropriate transaction and enters standby once again.

Should the Interrupt line be activated, the node performs a device sweep. This is done by individually addressing each device and making a confirm reading request. Addressed devices will transmit either a confirmation or denial. Once the requesting device is reached data is transmitted and the bus returns to standby status.

To avoid the possibility of conflicting requests, the device sweep will always address all devices before finishing. Should the General Interrupt line still be activated by then, another sweep will be performed. This choice of sweeping method presents advantages and disadvantages in different scenarios, concerning mostly reaction speed to devices with varying degrees of critical information. It does, however, completely prevent the scenario of devices being isolated by other devices with greater requesting frequency - such cases are most obvious when considering longer chains in a model where the sweep restarts from the first device after reaching a device with a reading; in this case, even devices with rather infrequent requests will cumulatively isolate devices further down the chain. It is also worthy of note that it is relatively easy to implement a priority queue, where the user can select the order of addressed devices; such a queue can even be automatically generated, for example through recommended priority scores.

8.3 Individual device control

8.3.1 Nodes

The state machine diagram of the typical node is as seen in figure 8.5.

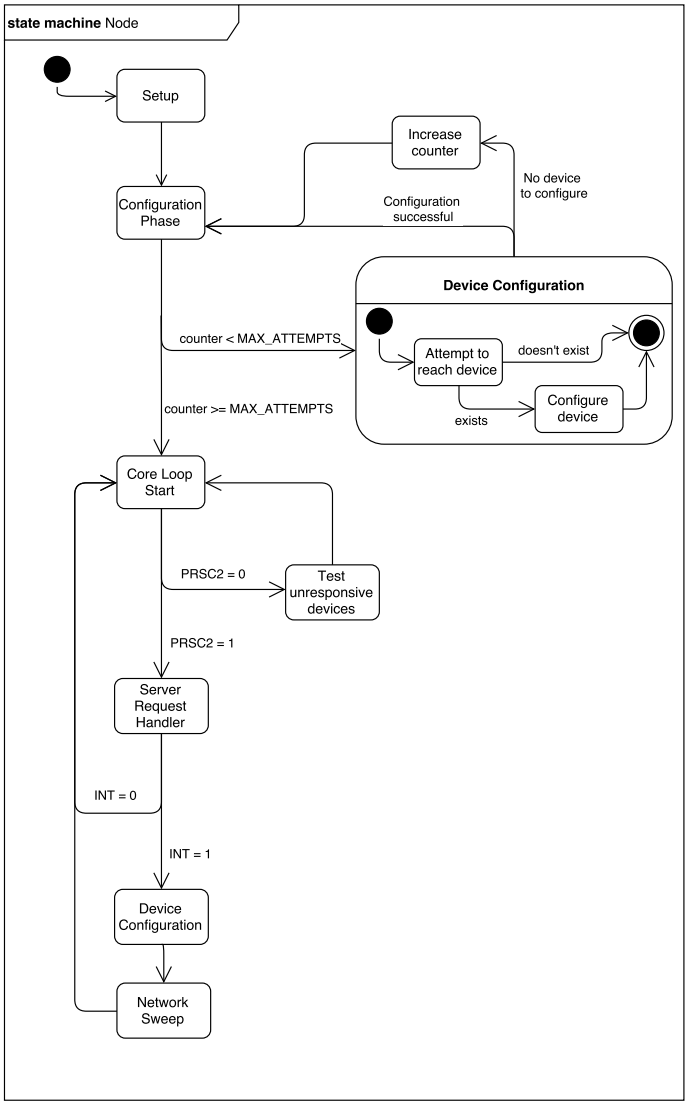

Figure 8.5: Node State Machine.

After starting all necessary internal systems, the node will proceed to search the network for configurable devices. This search has a set amount of maximum failures at system start-up, which continues if no device has been found until that point. After the initial search, and for as long as devices are connected to the network, the node will sleep until it is called by a device interrupt to perform a network sweep.

Network sweeps consist of a series of unicast requests from all present addresses, including the reserved address for new devices. This allows new devices to join the network even after the system is powered. Requests to configured devices are heavily dependent

on the characteristics of the device, but follow protocol standards and are detailed on the accompanying datasheet, obtained by the base station. Requests directly related to a device's functionality are documented in the associate file, along with a decision tree of available requests.

A device is considered unresponsive if its presence signal is low. However, since the presence lines only extend as far as adjacent members in the chain, slave devices are allowed to make an interrupt call to report a disconnected device. When the sweep call reaches the reporting device, the node starts prefacing reading calls with a status check; should the device be unresponsive, it is flagged as such and removed from the call queue. However, should all devices down the chain from the reporting device be unresponsive, the node considers they were physically removed, and releases their addresses to be taken by new devices.

The node, as the previous chain member to the first device, is also responsible for testing its presence signal. Nevertheless, while the detection method is simplified for the node, the testing and correction method is the same.

8.3.2 Slave Devices

The state machine diagram of the node is as seen in figure 8.6.

The setup phase of the slave device is critical for the configuration process. After preparing the input/output interface, the slave device waits for the configuration signal from the previous device to start its I2C interface. It's worthy to note that, before the token arrives, a device has its I2C interface disabled, being unable to interact with the bus at all. This enables the use of a single global address for not configured devices.

The core loop of the slave device has two main components: the function loop and the communication interface.

The function loop is a combination of predefined class methods present in the Function interface class, and the implementation of said functions by the developer. While the developer is free to add as many auxiliary functions as he or she intends, the main function loop must adhere to the conditions set by the system. These are, as of this prototype, the implementation of a non-void loop function that returns a non-zero value should the slave device request for the attention of the node. This allows the network interface to enable the Interrupt line, and prepare for the eventual request. Other conditions may also be set or recommended as the project develops; for example, while I2C communication supports long response times from the slave devices through clock stretching, these are discouraged since they impact the long term performance of the entire network.

The communication interface regularly checks the I2C interface for node requests. These are again largely dependent on the host system; for instance, the ATmega328 TWI assumes different states depending of whether the leased address was called for for receiving or transmitting, for example. In such a case, the instruction is routed to the instruction interpreter. The instruction interpreter is the second main point of interaction between the

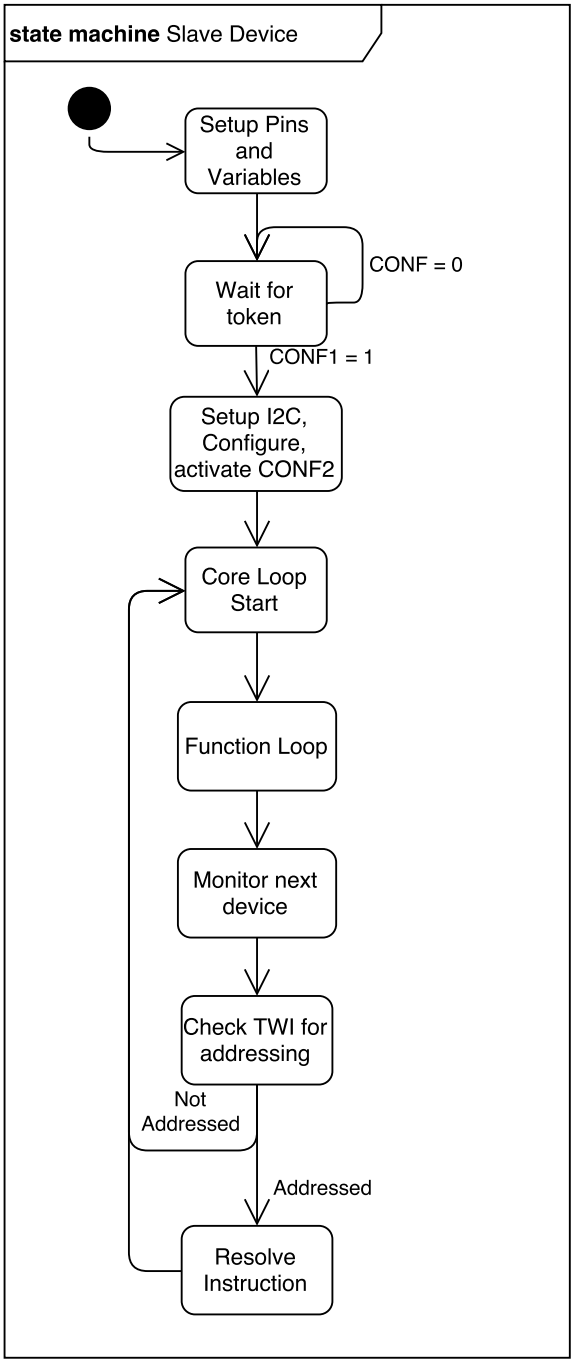

Figure 8.6: Slave Device State Machine.

configuration interface and the device function code. To prevent tampering and increase security, instructions with internal codes are parsed first through the core interpreter, and only if they are not recognized, that is, they are function specific codes, are they parsed by the function specific interpreter, where they can be directly routed to the specific routine requested by the instruction. From then on, the program is responsibility of the developer,

8.3.2.1 Temperature Sensor

The temperature sensor device has a very simple active type interface, that is, the interface makes requests from the node. This makes it very representative of the sensor type devices.

The device core loop consists of reading the temperature sensor and checking the value against the last recorded value. The method returns whether the device detected a change in temperature, and device control can then control the Interrupt Pin to make a node request.

The device instruction handler accepts a single instruction where the device returns to the node a one byte transmission with the sampled temperature value.

8.3.2.2 LED Controller

The LED controller device has a very simple passive type interface, where its function takes orders from the node but doesn't make requests through the Interrupt line.

The function has two main modes: a simple looping light pattern and discrete LED control. As such, the function core loop checks for the loop condition and runs it if true, or does nothing if false.

The device instruction handler has two instructions. One takes no parameters, and simply activates the looping light pattern. The second accepts a byte of parameters, and uses the 3 most significant bits to control each of the LEDs. Since this instruction naturally deactivates the looping pattern, it also conveniently serves as an off switch for the looping pattern.

Chapter 9

Conclusions

Contents

9.1 Conclusions 47
9.2 Contributions 48
9.3 Future Work 48

This chapter concludes this document, pondering on contributions, conclusions and future work.

9.1 Conclusions

From an analysis of the mainstream home automation market and the proponent's own interest, the context for this project quickly pointed to a niche in the home automation field - the budget, function oriented system that focused on security and efficiency.

The combination of initial system requirements for the envisioned home automation system presented a scenario where there weren't solutions that satisfied some of the greater needs of the system, namely the need for an interface that was both easy to use, modular and had the inherent security of wired systems. That proved to be an opportunity to develop a new framework for development of I2C-based self configuring networks.

A hardware based solution presented some relevant characteristics versus the more widespread software based solutions, like SMBus, allowing for more speed, simpler addresses and a bus overall better resembling I2C. Taking the context of the project, which expects multiple but simple devices, this simplification of the configuration phase was very desirable way to reduce complexity while maintaining ease of use.

The implemented prototype, which exemplifies the viability of this type of solution for automatically configured I2C based networks, is a very satisfying conclusion to this work, and presents a solid first step towards the success of this home automation project as a whole.

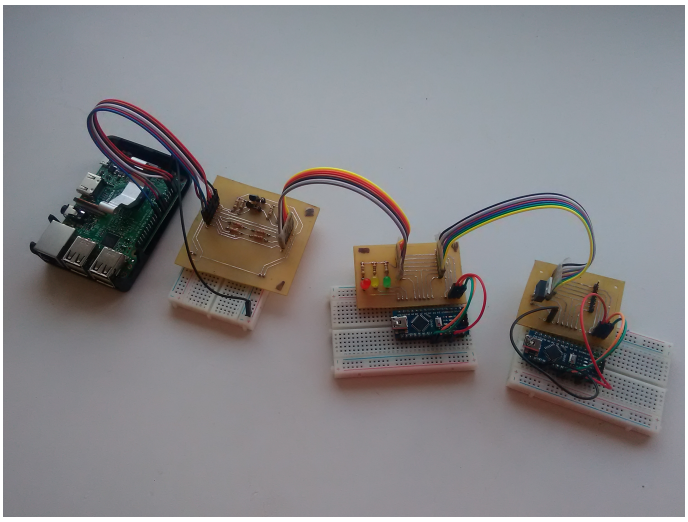

Figure 9.1: Developed prototype in chain arrangement; from left to right, Raspberry Pi, signal conditioning board, LED device and temperature sensor device.

9.2 Contributions

The specific design intentions of this system, such as the desired flexibility in performance, number and type of devices, cost and security, led to the conclusion that, while I2C based self configuration wasn't unheard of, current solutions presented limitations to the specific needs of this system design. Most, if not all, such solutions are heavily reliant of software control, which presents some restrictions that may not be appropriate to every system.

As such, it serves as a contribution the study made on various ways I2C based self configuration may be implemented, with a special focus on the hardware based ones, since these both happened to be more appropriate to this project specifically, and seemed to be in general less explored as solutions to this problem.

Additionally, following the context of this project, it serves as a contribution the project itself, as an answer to a need for an affordable, function oriented and scalable home automation system.

9.3 Future Work

Along with the context and motivations for this project, and as can be read in the system description in chapter 6, the system itself requires further work, such as:

- Implementation of the base station level of the system, including device management, user interface and configuration of devices in TEDS-style (as seen in IEEE1451);
- Development of various functional devices;
- Seamless vertical integration of all levels.

Appendix A

Probability calculation for duplicate tickets

As referred in section 5.3.1.1, regarding arbitration based enumeration with random number generation, the following table is presented with the expected probability of duplication of randomly generated identifiers, or tickets, given M devices and an N bit ticket. For this scenario, since the most atomic transmission is one byte, N considers only values that are multiples of eight.

Given the model for ticket collision, and *r=2*:

$$P(X \geq r) = \sum_{i=r}^{M} (\frac{1}{2^N})^i \times (1 - \frac{1}{2^N})^{(M-i)} \tag{A.1}$$

The results can be seen in table A.1.

Table A.1: Probability of ticket collision as per model A.1

	Ticket size (bit)		
Number of devices	8	16	24
1	0,00000000000%	0,00000000000%	0,00000000000%
2	0,00152587891%	0,00000002328%	0,00000000000%
3	0,00456571579%	0,00000006985%	0,00000000000%
4	0,00910765957%	0,00000013970%	0,00000000000%
5	0,01513992866%	0,00000023282%	0,00000000000%
6	0,02265081057%	0,00000034923%	0,00000000001%
7	0,03162866161%	0,00000048892%	0,00000000001%
8	0,04206190647%	0,00000065189%	0,00000000001%
9	0,05393903790%	0,00000083813%	0,00000000001%
10	0,06724861633%	0,00000104765%	0,00000000002%
20	0,27665998041%	0,00000442297%	0,00000000007%
30	0,61722176444%	0,00001012525%	0,00000000015%
40	1,07855757061%	0,00001815377%	0,00000000028%
50	1,65089433721%	0,00002850783%	0,00000000044%

Appendix B

Analysis of pull-up resistor value versus maximum chain size

As referred in section 7.2.1, I2C bus pull-up resistor size effectively limits the maximum number of attached devices in the bus.

Pull-up voltage, total bus capacitance, and intended bus speed are all factors in determining pull-up resistor sizing *Rp*. Since the maximum resistor size decreases as bus capacitance increases, it is reasonable to say that the maximum number of devices is achieved for the lowest value of Rp_{Max} that satisfies $Rp_{Max} > Rp_{Min}$.

The ATmega328 datasheet[26], as per table 32-10, page 372, states that capacitance contribution for each pin is 10pF, so for this exercise we simplify that total bus capacitance $C_b = N * 10pF$, where N is the number of devices, including the network master. The same page suggests formulas to calculate the maximum and minimum resistor values for combinations of pull-up voltage, total bus capacitance and bus speed. They are as such:

- Minimum pull-up resistor

$$Rp_{Min} = \frac{V_{CC} - 0.4}{3 \times 10^{-3}} \tag{B.1}$$

- Maximum pull-up resistor for $f_{SCL} \leq 100\text{kHz}$

$$Rp_{Max} = \frac{10^{-6}}{C_b} \tag{B.2}$$

- Maximum pull-up resistor for $f_{SCL} > 100\text{kHz}$

$$Rp_{Max} = \frac{3 \times 10^{-7}}{C_b} \tag{B.3}$$

This means that:

- Minimum pull-up values are $Rp_{Min} \approx 1.53\text{k}\Omega$ for $V_{CC} = 5\text{V}$ and $Rp_{Min} \approx 967\Omega$ for $V_{CC} = 3.3\text{V}$

- The maximum number of devices is given by

$$N = \frac{10^5}{Rp_{Max}} \tag{B.4}$$

for $f_{SCL} \leq 100\text{kHz}$ or

$$N = \frac{3 \times 10^4}{Rp_{Max}} \tag{B.5}$$

for $f_{SCL} > 100\text{kHz}$

Solving these minimization problems for the lower limits given by typical pull-up voltage values gives us the limits for device quantity for typical I2C configurations as in table B.1.

Table B.1: Maximum number of devices for typical I2C configuration values

	I2C bus speed	
Pull-up voltage	100kHz	400kHz
5V	65	19
3,3V	103	31

Any of these values is enough for typical home automation system, especially if we take into account that these networks are designed to handle single rooms. Some large office applications may require more devices, but appropriate configurations are also able to handle the expected number of devices.

Appendix C

System Classes

As referred in section 8.1.1, this appendix presents a more detailed overview of the system classes for the control software of the Level 2 bus[1]. The high level diagram of system classes is as seen in figure C.1.

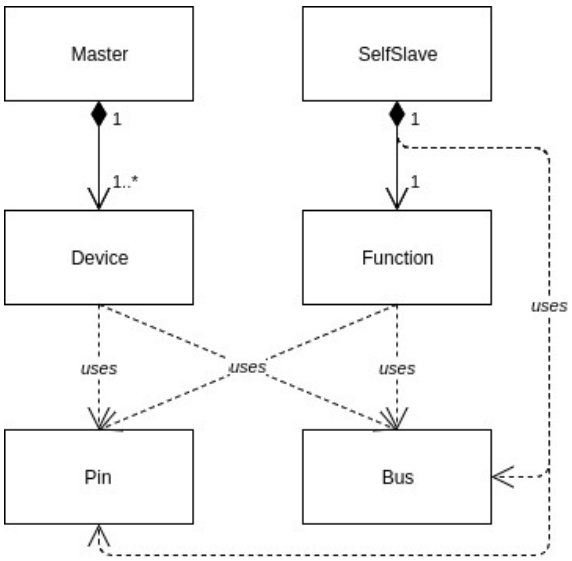

Figure C.1: System simplified class diagram.

With flexibility of the system as a main design goal, the development of proper interfaces for various platforms is key. Interfaces minimize work when porting the system to a new platform, and increase high level compatibility by offering a working template of desired functions to developers.

[1]Refer to figure 2.1.

C.1 Pin

The Pin class is mainly directed at system-specific input/output interfacing, since the project makes critical use of digital signals. This package has a consistent interface, so as to minimize platform specific work, but the methods are naturally tailored specifically for the host platform. For example, the ATmega328 micro-controller has a native library for input-output pin manipulation, and its library is quite simply a shell for those methods. On the other hand, Raspberry Pi uses a its pins in a write to file fashion, so the methods for its package are more developed in order to open, configure, manipulate and close the files optimally.

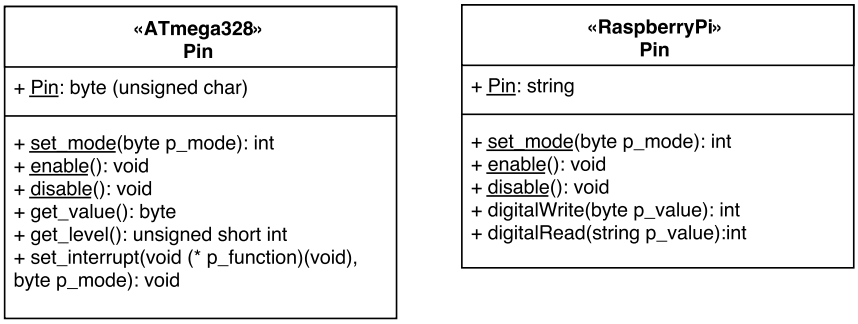

Figure C.2: Expanded Pin classes for ATmega328 (left) and Raspberry Pi (right), with common members and methods underlined.

C.2 Bus

The Bus class is directed to the I2C handling by the host system. Again, while the key interface members and methods are consistent across systems, auxiliary, atomic and other variables differ as the system needs. In this case, ATmega328 supports an approach to its I2C communication that allows for very strict, byte to byte control, so the support for variable length transmissions was custom made from those capabilities. On the other hand, Raspberry Pi, mainly due to its operating system, approaches I2C communications in a write to file fashion, allowing no more control than the choice of data buffer to transmit.

This class is also an acknowledgement of the eventual need of multiple I2C buses in the same host system, for example for interfacing with a sensor and the network at the same time.

C.3 Device

The Device class is an object oriented interface for the node to interact with slave devices. Objects are created dynamically as the slave devices join the network, and store critical device data such as current address, identification information such as manufacturer, model, serial number, and current status. The package contains the generic node side read

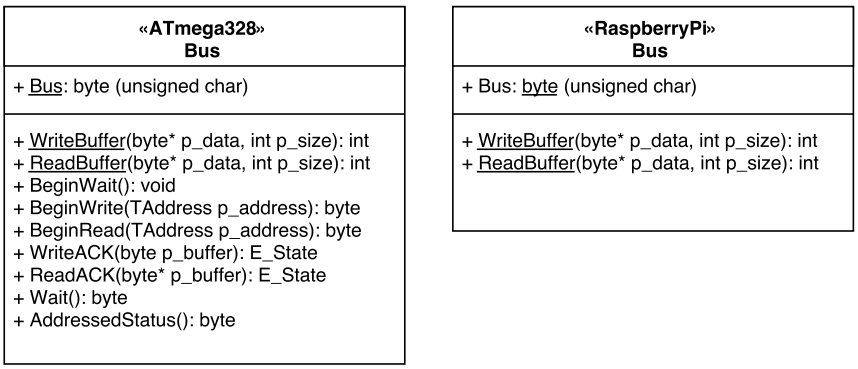

Figure C.3: Expanded Bus classes for ATmega328 (left) and Raspberry Pi (right), with common members and methods underlined.

and write methods, as per system protocol; additionally, several methods are included as shortcuts used to quickly access the standard protocol headers, named *instructions*, which are described in section 8.2.

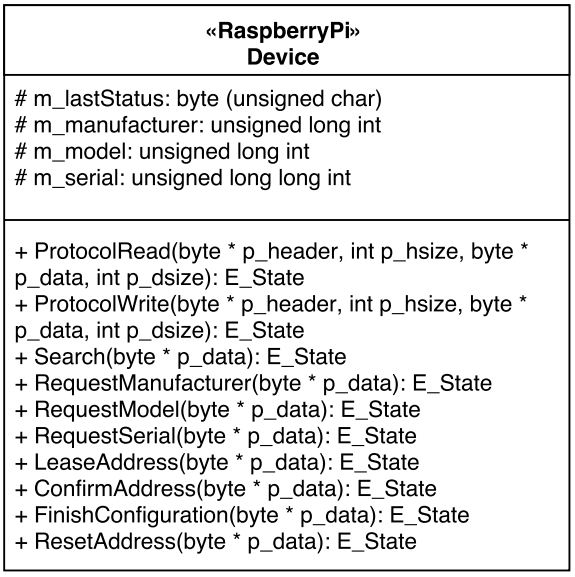

Figure C.4: Expanded Device class for the Node.

C.4 Master

The Master class contains the high level node functions, such as the core loop, network sweep, and, in this prototype, data processing. These high level processes are described in more detail further in chapter 8.

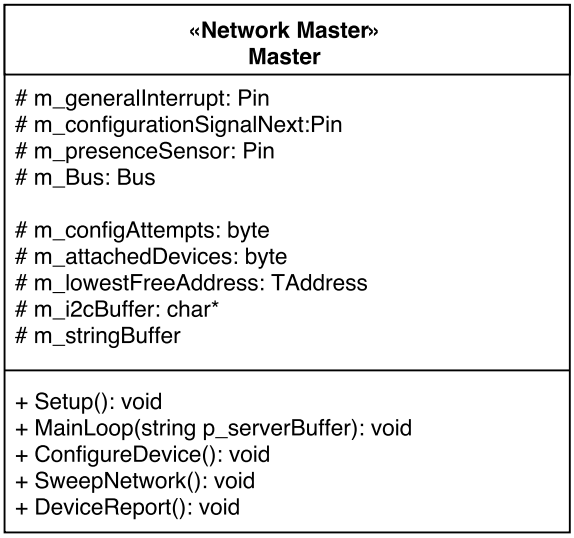

Figure C.5: Expanded Master class for the Node.

C.5 SelfSlave

The SelfSlave class contains high level slave device functions, such as core loop, instruction resolution and device self-management. In many ways this package is analog to the Master package, with some critical differences.

From the slave device's perspective, no device other than the node is relevant, or even required, in the network, and node information is not in such complexity that requires a dedicated class for handling. As such, this library does not have a Device equivalent for the node.

As per this system's protocol, the slave device may read from, or write to, the bus according to the transmitted instruction. For this reason, slave side protocol processing occurs in two phases, one of instruction reception and one of instruction resolution. Instruction resolution has a number of resolutions tending to configuration and other base tasks hidden from third party developers as a security and compatibility measure.

C.6 Function

The Function class contains the slave device function specific code and instructions. As such, specific algorithms, routines and actions are highly dependent on the implemented function. There is, however, a need for a standardized interface for this function to interact with the SelfSlave class. This is done with two main class methods - one for the function specific loop, and one for the function specific instructions and instruction resolution.

The contents of these methods may vary, as different functions have different needs. For instance, an event based, change-detecting sensor must have a core loop of sampling its respective parameter and comparing to previous values, while its instructions could be

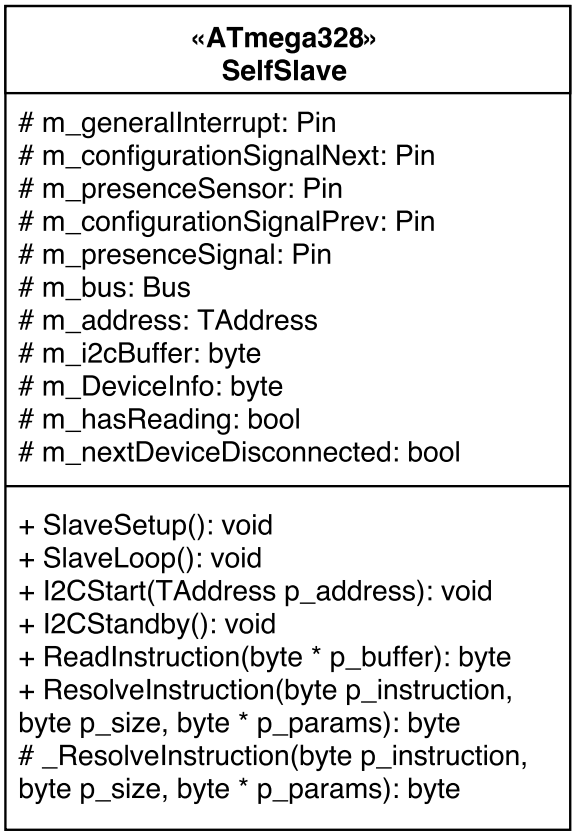

Figure C.6: Expanded SelfSlave class for the Slave Device.

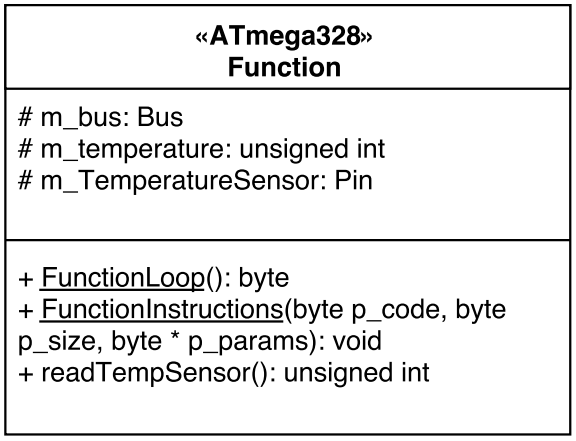

Figure C.7: Expanded Function class for the Temperature Sensor Slave Device. Underlined methods are required by interface.

as little as one transaction that communicates all sampled values. On the other hand, actuators will have very simple or non-existant loop cycles, and resolve all their tasks on instruction handling. For this, pre-declared function handlers are standardized, and their code can be constructed as needed.

References

[1] Smarter Home Life. Google formally enters the home automation market with google home. https://smarterhomelife.com/everything/2016/10/4/google-formally-enters-the-smarter-home-market-with-google-home, 2016.

[2] Google Inc. Google home overview page. https://madeby.google.com/home/, 2016.

[3] Amazon.com Inc. Amazon echo overview page. https://www.amazon.com/dp/B00X4WHP5E, 2016.

[4] Clipsal Australia Pty Ltd. C-bus lighting control via the c-bus pci quick start guide. http://training.clipsal.com/downloads/OpenCBus/C-Bus%20Quick%20Start%20Guide.pdf, 2008.

[5] Zigbee Alliance. Official zigbee homepage. http://www.zigbee.org/.

[6] Insteon. Official insteon homepage. http://www.insteon.com/.

[7] KNX Association. What is knx? https://www.knx.org/knx-en/knx/association/what-is-knx/index.php.

[8] KNX Association. About knx members. https://www.knx.org/ae/community/manufacturers/about/index.php.

[9] ICONTROL. 2015 state of the smart home report, 2015.

[10] Business Insider UK. Google's parent company is deliberately disabling some of its customers' old smart-home devices. http://uk.businessinsider.com/googles-nest-closing-smart-home-company-revolv-bricking-devices-2016-4, 2016.

[11] Stephen Cobb. 10 things to know about the october 21 iot ddos attacks. https://www.welivesecurity.com/2016/10/24/10-things-know-october-21-iot-ddos-attacks/, 2016.

[12] NXP Semiconductors. *UM10204: I2C-bus specification and user manual*, 2014.

[13] Jean-Marc Irazabal and Steve Blozis. *Application Note AN10216-01: I2C Manual*, 2003.

[14] Maxim Integrated. Application note 476: Comparing the i²c bus to the smbus.

[15] James Lyke et al. A plug-and-play approach based on the i2c standard. In *24th Annual AIAA/USU Conference on Small Satellites*, 2011.

[16] Motorola Inc. *SPI Block Guide*, 2003.

[17] Maxim Integrated Products Inc. Application note 3947: Daisy-chaining spi devices.

[18] Robert Johnson, Kang Lee, James Wiczer, and Stan Woods. A standard smart transducer interface - ieee 1451. In *Sensors Expo, Philadelphia*, 2001.

[19] National Instruments Corporation. *An Overview of IEEE 1451.4 Transducer Electronic Data Sheets (TEDS).*

[20] Internet Systems Consortium. *ISC DHCP*, 2014.

[21] European Union Agency for Network and Information Security ENISA. Major ddos attacks involving iot devices. `https://www.welivesecurity.com/2016/10/24/10-things-know-october-21-iot-ddos-attacks/`, 2016.

[22] Maxim Integrated. 1-wire overview page. `https://www.maximintegrated.com/en/products/digital/one-wire.html`.

[23] SBS Implementers Forum. *System Management Bus (SMBus) Specification*, 2000.

[24] Texas Instruments. *LM35 Precision Centigrade Temperature Sensors*, 2016.

[25] LLC Semiconductor Component Industries. *BS170G Small Signal MOSFET Datasheet*, 2011.

[26] Atmel Corporation. *ATmega328/P Datasheet*, 2016.